Fast And Space Efficient Trie Searches

Phil Bagwell

Searching and traversing m-way trees using tries is a well known and broadly used technique. Three algorithms are presented, two have constant insert, search or delete cost, are faster than Hash Trees and can be searched twice as quickly as Ternary Search Trees (TST). The third has a lgN byte compare cost, like a TST, but is faster. All require 60% less memory space per node than TST and, unlike Hash Trees, are smoothly extensible and support sorted order functions. The new algorithms defining Array Compacted Trees (ACT), Array Mapped Trees (AMT), Unary Search Trees (UST) and their variants are discussed. These new search trees are applicable in many diverse areas such as high performance IP routers, symbol tables, lexicon scanners and FSA state tables.

Categories and Subject Descriptors: H.4.m [**Information Systems**]: Miscellaneous

General Terms: Searching,Database

Additional Key Words and Phrases: Trie,Tries,Search,Trees,TST,ACT,AMT,UST,Symbol Table

1. INTRODUCTION

The concept of tries was originally conceived by Brandais [1959] and later given that name by Fredkin [1960] which he derived from the word *re***trie***val*, in *information retrieval systems*. Tries are one of the most general-purpose data structures used in computer science today, their wide application makes any performance improvement very desirable.

In this paper I present two algorithms, together with variants, that for a set of N keys have a cost, for inserts, searches and traversal iterations, that is independent of N. The third has a lgN byte compare search cost, like the elegant Ternary Search Trees (TST) by Bentley and Sedgewick [1997]. These algorithms were originally conceived as the basis for fast scanners in language parsers where the essence of lexicon recognition is to associate a sequence of symbols with a desired semantic. M-way tries have been shown to be a powerful technique to create this association and many hybrid algorithms have been developed, examples include Hash Tries, Bentley, McIlory, and Knuth [1986] and TST.

The method for searching tries is conceptually simple. It can be illustrated by finding whether the key **ADD** exists in a map, perhaps an entry in a symbol table,

Address: Es Grands Champs, 1195-Dully, Switzerland

organized as an m-way trie. Start at the root node, take the first letter of the key, **A** and use its ordinal value to index down the branches to the corresponding sub-trie. Next take the **D** and find the subsequent sub-trie. Finally take the last **D** and find the terminal node that will contain the value to be associated with the word **ADD**. If at any stage a sub-trie cannot be found to match the particular character then the key is not in the symbol table. A partial tree is depicted in figure 1.

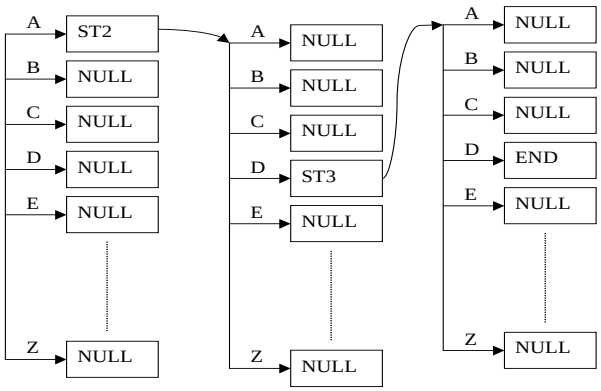

Fig. 1. An Array Tree.

Direct table indexing represents the fastest way to find a sub-trie, however all the sub-tries must be created as branch tables with a length equivalent to the cardinality of the symbol set. Each possible symbol is allocated a place in the table according to its equivalent ordinal value, the entry contains either a pointer to the next sub-trie or a null value to indicate a non-member.

Given this structure then the simple C++ code fragment, Figure 2, can be constructed to enable keys to be found in a time independent of the number of keys in the map.

```
// Assuming characters are represented as A=0, B=1,...,Z=25
class Table{Table *Base};
  Table *ITable;
  char *pKey; // pKey points to the zero terminated Key string
  ITable=RootTable;
  while((*pKey)&&(ITable=ITable[*pKey++]));
```

Fig. 2. An Array Tree Search Code Fragment.

The drawback to this approach is only too apparent. The space required for tables is proportional to s^p where s is the symbol alphabet cardinality, 26 in the example above, and p is the length of the keys to be stored in the map. For 5 letter keys it could require 26^5 or about 12 million nodes.

This concept has been known since the early 60's as has the associated problem of the high space overhead for null-pointers and for this reason the approach has

usually been discarded as impractical, Knuth [1998] pages 492-512. As an alternative compression techniques have been used within hybrid trees to give better packing of the pointers but suffer attendant time penalties. In this paper I exploit the properties of key sets to create new data structures that minimize these difficulties.

In typical key sets the number of trie branches per trie is small. For a random set of N keys from an alphabet M, the branches at each level l of iteration of the search is given by $M^l(1-(1-M^{-l})^N) - N(1-M^{-l})^{N-1}$, Knuth [1998] page 725.

With randomly ordered and most typical key sets early branches tend to the cardinality of the alphabet, while later tries tend to be sparse. However, many useful key sets contain a common or fixed prefix which lead to fuller tries towards the end of the key sequence. This branch count distribution can be used to achieve an advantageous balance of speed against memory use.

The general problem of taking an m-way branch in a trie can be encapsulated in a function I will call *Next(Node,Symbol)*. It takes as arguments the current tree node and the symbol value indicating which sub-trie or arc to follow. The function returns the sub-trie node or a null value if there is no valid branch. The challenge is to implement this function efficiently. This form is immediately recognizable as an FSA table transition function.

```
// Generic Key Search in a trie based Dictionary
int Find(char *Key)
  {
  TNode *P,*C;
  P=Root;
  while(C=Next(P,*Key)){Key++;P=C;}
  if(*Key==0)return P->Value;
  return 0;
  }
```

Fig. 3. A Generic Trie Search Function.

Given this function, Figure 3 shows an algorithm to find a zero terminated string key in such a tree. This is a very compact and efficient search method, however the central problem has just been delegated to the *Next* function, the question remains as to how well this can be implemented. A simple and fast approach is to use the TST.

1.1 Ternary Search Trees (TST)

A TST is a hybrid, combining the concept of an m-way tree and a small binary tree to find the sub-trie needed to match a specific symbol. Figure 4 illustrates the structure; as with the previous example start at the root but, instead of direct indexing an array a small binary tree is searched to find the sub-trie corresponding to the **A**. Once done, the **D** is taken and the search continues in the next binary tree associated with this sub-tree and finally repeated for the ending **D**.

This approach solves the wasted space problem and the binary search proceeds very quickly, as only one character is compared rather than a full key as would be

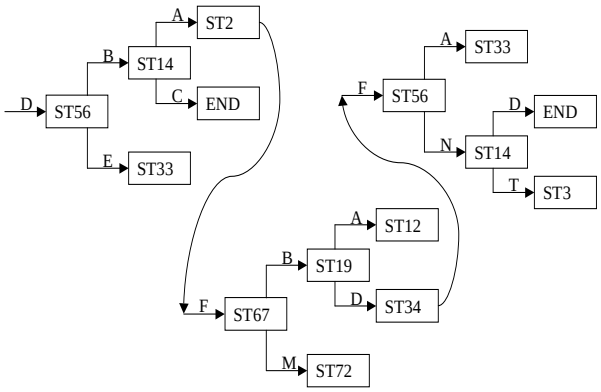

Fig. 4. A Ternary Search Tree.

the case in a simple binary search tree. Figure 5 shows an implementation of the *Next* function using a TST.

In the published form each TST node requires three pointers, a split character and the associated value. Clement, Flajolet, and Vallee [1998] have shown that the cost of the algorithm is proportional to lgN byte comparisons where N is the number of keys in the dictionary. In practice a TST seems to perform better than this would suggest, the cached memory hierarchy of modern computers and the skew in the distribution tree branches give enhanced performance.

```
// TST Next Method
Class Node {
  char Splitchar;
  Node *equal,*kidl,*kidr;
  };
inline Node *Next(Node *P,unsigned char c)
  {
  do{
    if(P->Splitchar==c)return P->equal;
    if(P->Splitchar>c)P=P->kidl;
    else P=P->kidr;
    }while(P);
  return NULL;
    }
```

Fig. 5. The *Next* function based on a TST

Modern computers incorporate one or more caches between the CPU and main memory, frequently accessed locations and algorithm instructions tend to be cache resident. The branch distribution for tries towards the root of the tree tend to the cardinality of the alphabet and, because they are close to the root, are most frequently accessed. The majority of the nodes in the early part of the tree will

therefore become cache resident. In many machines the cache functions at between ten and twenty times the speed of the main memory, hence the binary search at the root of the tree runs much faster than when accesses are being made randomly into main memory further down the tree. For a moderate size dictionary, up to 50000 keys, the performance of a TST is excellent and almost independent of the number of keys. However, for very large dictionaries or where main memory performance approaches the cache speed or where the dictionary is only infrequently referenced from a larger application then the lgN performance becomes apparent.

The performance of a TST, as with simple binary trees, can be degraded by the same degenerate case of inserting keys in order, instead of the benefits of a lgN search at each trie branch it can degenerate to an $N/2$ search, where N in this case has a maximum of the alphabet cardinality. For the performance comparisons, tree balancing was added to the TST insert function. The performance of this enhanced version, TST*, is reported in Table 3 towards the end of the paper.

1.2 Array tries

In implementing the *Next* function the core problem is to eliminate the space lost with null-pointers while retaining the speed of direct indexing. The algorithms for Array Compacted Trees (ACTs) and Array Mapped Trees (AMTs) both do use direct array indexing and have a search time independent of the dictionary size. ACT is far more complex but does give a search algorithm that is close to ideal. AMT on the other hand is a robust, less complex, hybrid algorithm with excellent performance.

2. ARRAY COMPACTED TREE (ACT)

Most branch tables are sparsely filled, therefore by careful arrangement two or more tries may be overlaid in the same space, Knuth [1998] page 493. An example, illustrated in table 1, will demonstrate the principles involved, a dictionary containing the keys **ABLE**, **BASE**, **BEAR**, **BANE** and **BIN** will be searched for the existence of the keys **BEAR** and **BEG**.

In the illustration the table represents the nodes in a trie based dictionary. The letters in the alphabet A to Z are associated with the ordinal values 0 to 25. Each node has a node number in the column headed *Node* and two pointers to other nodes *Parent* and *Xbase*. The *Parent* address is used to mark each of its *Child* nodes. On indexing into the trie if the node marker matches the *Parent* address then the character is a member of the trie, otherwise it is not. *Xbase* points to the base of the next sub-trie. The column headed *Letter* is not part of the node data but has been added to make the example easier to follow.

Now test for the existence of the key **BEAR** in the dictionary. The first letter in **BEAR** is **B**. Start at the root node at node 0 the *Xbase* for this trie is 1 and is added to 1, the value of **B** to arrive at node 2. Notice the letter **B** appears in the column headed *Letter* and verifies that the procedure has been followed correctly. Its *Parent* is 0 indicating that this letter is indeed a member of this trie.

The new *Xbase* is 4, now add 4, the value of **E** to give 8. The *Parent* node for node 8 is indeed 2 so **E** is a member of this trie. The new *Xbase* is 9 to which 0 is added, the value of **A** to give 9. The *Parent* node for node 9 is 8 so **A** is valid. Next take the value of **R**, 17 and add to the next *Xbase* of 1, giving node 18. The

Parent node is 9 so this is a valid branch too.

All four letters where found while traversing the trie thus **BEAR** is in the dictionary.

Table 1. Overlaid tries.

Node	Letter	Parent	XBase	Node	Letter	Parent	XBase
0		NULL	1	10			
1	A	0	2	11			
2	B	0	4	12	I	2	3
3	B	1	2	13	L	3	1
4	A	2	1	14	N	4	2
5	E	13	NULL	15			
6	E	14	NULL	16	N	12	NULL
7	E	19	NULL	17			
8	E	2	9	18	R	9	NULL
9	A	8	1	19	S	4	3

Now repeating the exercise with **BEG**, all goes well for **B** and **E** to arrive at node 8. Next take the **G**, value 6, and add it to the *Xbase* of 9 to give 15. The *Parent* of node 15 is 12 and not 8, there is no match for the **G** and hence the word **BEG** is not in the dictionary.

In inserting the tries it is essential to ensure that no valid branches collide and occupy the same location.

With this concept of overlaid tries a very lean implementation of the *Next* function can be realized. Figure 6 contains a suitable class declaration and the *Next* function. As with the previous implementation using a TST, the function returns a pointer to the next sub-trie if the character is a member of the trie or a NULL if it is not. Utilizing this *Next* function with the *Find* function in Figure 3 it can be seen, by inspection, that the resulting algorithm takes a small constant time for each key character that is independent of the total number of keys.

```
class ACTSNode {
  public:
    ACTSNode *Parent,*IndexBase;
    unsigned char EndZone:2,Term:1,Free:1,
    FirstChar,NextChar,BranchCnt;
    int Value;
    }
  ACTSNode *Next(Node *P,unsigned char c)
  {
    ACTSNode *C;
    If((BranchCnt)&&(((C=P->IndexBase+c)->Parent)==P))return C;
    else return NULL;
  }
};
```

Fig. 6. The *Next* function based on an ACT

Note that the last sub-trie is detected when a node has a zero branch count. This algorithm does indeed approach the ideal, is not dependent on the dictionary size and moreover gives the minimum number of memory accesses. The goal of achieving the speed of direct array indexing has been reached, but apparently at the expense of having a difficult insertion problem to solve. If the algorithm is to be space efficient a variation on the familiar *knapsack packing* theme must be solved which could be computationally expensive.

Favorable trie branch distribution allows an insert algorithm to be created that has a comparable cost to other trie insert algorithms, without this ACT would remain a curiosity confined to a few special applications where dictionaries are static, such as spelling checkers, router tables, and so on.

2.1 Packing The Knapsack

The order of difficulty in packing tries into memory can be perceived by selecting a couple of dozen words and to do the task by hand. First define the trie entries and attempt to overlay them in the smallest space. Starting with the tries with most entries first, it is surprising that it is not such a difficult job after all

It is soon discovered that there are a few tries with many entries and many tries with only one or a few entries. With N random keys made up from s symbols then on average after $\log_s N$ symbols keys are unique. For example, taking 18000 random keys composed from the 26 letters A to Z this happens after about three letters. Tries after the 3^{rd} letter will frequently have just one entry.

The root trie will often be fully filled, each entry having 18000/26 ways to be filled. The tries for the second letter in the key will most likely be filled, there are $18000/(26 * 26)$ ways for this to happen. By the third letter the tries will start to become more sparsely populated. From the forth position onward there will be only a few entries.

Table 2 shows the distribution of tries by number of branches for an ACT constructed from an English language dictionary containing 149000 unique words. The alphabet includes the letters A to Z and a few special characters. Notice how quickly the number of tries drop with increasing branch count.

Table 2. Tries with a given number of branches.

Branches	**Tries**	**Branches**	**Tries**	**Branches**	**Tries**
1	103027	11	165	21	14
2	190192	12	114	22	6
3	34500	13	59	23	17
4	12423	14	64	24	7
5	4548	15	60	25	5
6	2131	16	40	26	7
7	997	17	39	27	5
8	567	18	21	28	3
9	363	19	29	29	1
10	269	20	11	30	1

It is this distribution that makes the puzzle solvable. Many of the tries to be inserted will be small and unconstrained. View the problem as fitting a few large

objects and a lot of "sand" into the knapsack, the higher the ratio between sand and larger objects the easier it is to fit them all in. Adding one more object becomes easy, just push some sand out of the way.

Unlike the pencil and paper example earlier, most real world problems do not give one the luxury of knowing all the keys before starting to create the key dictionary. As the number of keys is not known in advance therefore it is not possible to start with the largest and most constrained tries first. Each time a new key is added one of the tries must be modified with the consequence that any individual trie may grow unpredictably.

2.2 Memory Management

Before planning the actual insertion for trie nodes, the general memory management strategy must be considered. As was seen in the first example, individual ACT nodes cannot simply be taken from the heap. The location of each node needs to be specifically controlled so that it occupies the correct offset from an index base.

Special care must also be taken at the end zones of the storage area. Recall that the search detects non-valid characters by the corresponding child node in the trie having a different parent. If the trie base is located closer than the cardinal value of the alphabet to the upper end of memory then an illegal memory reference would occur for any characters causing an access beyond this boundary. This bounds check is made at insert time thus removing the overhead for all searches. The alternative, using wrap-around or a modulus technique, was discarded for this reason.

ACT nodes are therefore allocated in contiguous blocks each time more space is required. This size is set to increase total node space by some factor, say 10 percent. However, the minimum size block allocated must be larger than the alphabet cardinality number of nodes. Empirically I have found that a block size greater than sixteen times the cardinality gives good performance, hence for an alphabet of 256 characters, greater than 4096 nodes are allocated at a time. The more memory allocated in each block the easier the packing task becomes.

2.3 Placing Tries

The very first and subsequent tries nodes must be located to minimize collisions. Each time a trie node from one trie collides with one from another, one of the tries must be relocated, an expense best avoided. One alternative is to use a random number generator to choose a location. This is a reasonable solution, but it is not the fastest, even at moderate space occupancy the time wasted on collisions and retries leads to a poor insertion performance.

Analysis of the node branch frequency suggests a better approach. Clearly it is best to distribute the nodes with the most branches at least the alphabet cardinality number of nodes apart. In this way they would have no risk of collision. This would be theoretically possible when the complete set of tries is known, but would not be the case for progressive insertion and extensible tries. However, recall that the most accessed tries tend to be the ones that eventually have the most branches and that the tries most accessed tend to be defined earliest in the creation of a dictionary. For example, with random keys, the root trie will be the first started and typically be the first to fill.

The strategy that proves to be most successful is to use a binary allocation. The

first trie base created is allocated at the half way free storage point. The next at the quarter array size point, the next at the three quarters, the next at the one eighth and so one. The earliest placed tries tend to grow the most and are spaced well apart. The smaller tries tend to be added later and fill the gaps.

Knowing this allocation pattern the process can be enhanced by initially linking the free nodes in this distribution order. New tries are created by drawing a new node from the free list with the knowledge that they will be well spaced from their neighbors.

This interlaced row sequence can be generated in two ways. One uses a binary counter and reverses the bit order, each new location is defined by incrementing the counter and then exchanging the left and right bits to reverse the bit order. The other generates the sequence directly using a mirror counter, the highest bit position is incremented and instead of allowing carries to propagate left they are propagated right, the carry is added to the next lower bit, rather than to the next higher one. The sequence is the mirror of that produced by simply incrementing a counter. When a new block of ACT memory is allocated all the new nodes are linked in a free list using the mirror counter to create the order. On allocation a new trie will automatically be placed with a low risk of collision. Tests have shown this method to be four times faster than using a simple random insert.

2.4 Growing Tries

Nearly every time a new key is added to the dictionary a character will be added to an existing trie. This will only fail to happen when the new key is a prefix of an existing key. Often this new location will be free and the insertion is simply a matter of removing the free node from the free memory list and creating the new trie entry. If the new location has already been used there is a high probability that it will be by a trie with only one entry. This type of trie can be relocated to the first available free node to make room for the entry to be inserted on the trie being extended.

Occasionally, the new entry will collide with nodes of another multiple entry trie, the only recourse is to entirely move one of the two tries to another location in memory. To keep the problem as simple as possible the tries are compared for length and the shortest moved. The short one is the easiest to fit elsewhere. There are a few special cases that cause exceptions to this rule.

2.5 Moving Tries

Finding a new location for a trie is the most complex operation in ACT's. A trie can be imagined to be like a comb with teeth in the places where there are entries and all the other teeth broken off. The task then is to try finding a place in memory where the *teeth* match up with nodes that are either free or are easy to move. The *comb* is systematically tested in different locations until a match is found. Typically the location is found after only a few attempts yet occasionally it proves to be impossible. The free node list may be close to exhaustion for example. The solution that bounds the insert time is to extend the free space by allocating another set of ACT nodes using more of the system heap. The limit for the number of retries sets the upper bound on the insertion time and can be traded against memory efficiency.

An implementation of an ACT based on the concepts so far presented performs well enough to be of real practical use. Search speed is better than in either a Hash Tree or a TST. The insert speed will be comparable to Binary Trees but space is not efficient, about fifteen to twenty percent of the free nodes will be wasted. This may not be of concern in some applications, but in others space use may be critical.

This memory waste is caused by the forced allocations needed when tries cannot be found new homes. With short keys and relatively well filled key spaces the tries tend to become well filled, progressively more become candidates for additional memory allocation. A significant reduction in unused nodes can be made by using one of the knapsack packing strategies, namely constraint reduction, Brown, Baker, and Katseff [1982]. If there are several objects already in the knapsack and another large irregularly shaped object needs to be added, then move some smaller ones out the way, these being easier to pack because they are less constrained.

Memory utilization can be improved significantly by applying this constraint reduction thus allowing a large trie to be inserted by displacing one or more small tries. A small trie is less constrained than a large one. It is simpler to find a place for 3 small tries with 3 branches than one large trie with 8 branches. The overall insert time is slightly reduced and, for typical key sets, the wasted memory eliminated almost completely.

As a result of these optimizations and the smaller node data structure, ACT uses less memory space to represent a dictionary than standard Hash Tree, Binary Tree or TST implementations. However, an ACT is more complex to code.

2.6 ACT's Data Structure

The data structure of the ACT is the key determinant of memory use and performance. The smaller this structure's size the better. But it is a compromise, a balance between the information needed to allow efficient key insertion and the memory used. In a 32 bit address model an ACT may be implemented with 8, 12 or 16 bytes per node.

An ACT can be implemented for any symbol representation bit size. Practical constraints limit the useful range from 2 to 16 bits. An ACT uses two address pointers, Parent and Index Base, up to 4 bytes of node control data and a value or pointer to associate with the key. A suitable class definition is shown in Figure 7.

```
class ACTSNode {
public:
  unsigned char EndZone:2,Free:1,Term:1,
    StrCnt:4,FirstChar,NextChar,BranchCnt;
  int Value;
  ACTSNode *ParentA;
  ACTSNode *IndexBaseA;
  };
```

Fig. 7. The ACT data structure.

The control data bytes are not used at all in the search function but are used during insertion to speed up adding symbols to tries and to enable fast fit tests

when tries require relocation. The control bytes also enable the fast traversal or sequential access to keys stored in the dictionary.

When colliding tries need to be moved new bases must be chosen for the trie and tested until a fit is found. Each of the character entry positions in the trie must be compared with the corresponding positions in a new potential location. If characters are selected from an alphabet with cardinality s then each of the s entries need to be checked for collision. However, as most of the tries contain only a few branch entries, this would be costly. A linear linked list overcomes this sparse array problem. The number of entries is limited to the symbol set cardinality hence where $s = 256$ only one byte is required to represent each link, for larger cardinalities this size must increase.

The head of the list is carried in a byte called *FirstChar*, it contains the character value of the first character in its trie. Each of the trie child nodes carries a forward link to the next character in the trie called *NextChar*. The end of the list is marked by a zero.

For the vast majority of key collections this approach works extremely well, most tries contain only a few entries so scanning through the list sequentially is quickly done. However, there are some useful key collections that lead to the tries being more complete, an example would be keys issued sequentially in blocks such as manufacturing part numbers. In these cases the lists become long and the scanning time becomes lengthy. This apparent problem of completeness leads to an optimization that gives a speed improvement.

Each time an entry is added to a trie then the link list grows by one. But at the same time, if the entries are distributed randomly the distances between entries in the trie decreases. There is a point at which it will cost less to directly index into the trie entries and search linearly backwards through the array to find a previous entry rather than search down the linked list. If there are p entries in a trie of cardinality s then the average distance between entries will be s/p and the cross over point is defined when the average search costs are equal, that is when $(s/n)/2 = n/2$ or when $n = \sqrt{s}$. However, empirical tests show that the algorithm to search backwards is approximately twice as fast as searching forward in the linked list and the cross over point then becomes when $n = \sqrt{s/2}$. Further, typical key sets tend to cluster around a sub-set of the alphabet allowing the cardinality s to be substituted by the actual alphabet range used.

The insertion algorithm chooses the best strategy to follow based on the number of entries already in the trie, typically reducing insert time by 4%. The entry count is available as the *BranchCnt* in the parent trie Node.

The linked list provides the structure to support rapid sequential key access and selective key searching.

2.7 Adding New Nodes to a Trie

When adding a new node to a trie there are three cases to consider.

- The node is free. The free node is removed from the free list and the extra branch node is created.
- The colliding node has a single branch. The colliding node is relocated to the first available free node on the free list.

- The colliding node is a member of another multi branch trie. The smaller of the tries is relocated.

The first two cases are most common. For the third case, several approaches have been assessed with the goal of minimizing the search time needed to find a suitable new location. The search is guided by the control information in the node, the *Free* flag, *BranchCnt* and *StrCnt* for the tail compressed string version.

The trie to be relocated is tested for a fit at the next and subsequent free nodes on the free list until an entry point is found, the trie is then relocated and parent/child pointers are updated. Care must be taken to ensure that special cases are dealt with correctly, for example, a child node being moved to create space may be the parent of the trie being inserted.

After a number of attempts to fit a trie the effort is abandoned, a new block of free nodes added, linked to the free list and the trie fitted in this new space. This sets a bound on the maximum time for an insert, but at the expense of less efficient memory utilization. Unlike hash tables, the structure is dynamic and extensible. As mentioned earlier, the growth granularity is made a function of the current size of the tree and the alphabet cardinality.

When keys are inserted into an ACT the node that corresponds to the last character in the key is marked as the terminating node by setting *Term* true. For the same reason partial match, nearest match and range checking are easily constructed. Note that arbitrary character sort sequences can also be obtained without recreating the structure, a useful feature.

2.8 Squeezing Memory Use

The basic ACT uses four words in a 32 bit architecture implementation, or 16 bytes. This can be reduced by increasing complexity or by accepting a slower insertion speed. Reduction to 12 bytes can be achieved after realizing that the last node in a key has no sub-trie index base and this space may be used to store the associated key value. However, any keys that are a prefix of another key seem to require exclusion. There are applications where this is acceptable, routing an IP address being one example, though in general it would limit the usefulness of the algorithm. However, the prefix termination and the associated value can be stored with a fixed offset to the trie node terminating the prefix. *Term* is set True to indicate a prefix, the cardinality of the alphabet added to the offset and the associated key value stored in this *ghost* node, essentially doubling the cardinality of alphabet and increasing the insert overhead marginally. Apart from this inconvenience the node size is reduced by one pointer and the search performance improves.

In some applications insert time is not critical but minimizing space may be. In these special cases a little trickery can be used to save the 4 control bytes and reduce the footprint to 8 bytes per node. The *First* and *Next* character bytes are removed requiring that during trie moves the memory must be scanned from the trie base to its upper limit to find all the entries and the branch count computed. The insert speed suffers as a direct consequence, for alphabets with small cardinality, up to 50, this is only a few percent, however for a cardinality of 200 or more this can be 3 times slower than the standard version.

Memory is allocated ensuring Node pointers are aligned to even word boundaries.

The *BranchCnt* reduced to just two bits, now only indicating zero, one, few or many trie entries, together with the *EndZone* markers, *Term* Flag and *Free* Flag are stored in the bottom 3 bits of the *IndexBase* and *Parent* pointers.

In this form the tree uses 60% less memory than the published version of the TST, which uses 20 bytes per node.

Similar techniques can be applied to AMT and UST, which follow, and the TST described earlier.

3. ARRAY MAPPED TREE

Now I will describe a third way to eliminate the null pointers in a trie, this is called an Array Mapped Tree. The AMT is another hybrid tree yet like the ACT search time is independent of the number of keys in a dictionary. The AMT is simple to code and has a search and insert performance that is comparable to an ACT.

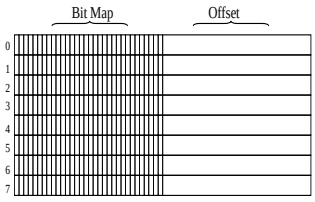

Fig. 8. The Bit Map Structure for an AMT.

The core concept is to use one bit in a bit map for each node that represents a valid branch in the trie. For an alphabet cardinality of 256 this will require 256 bits The symbol is used to index to the associated bit in the bit map. This bit is set to one if a sub-trie exists or zero if there is none. A pointer list is associated with this bit map containing an entry for each one bit in the bit map. Finding the branch for a symbol s, requires finding its corresponding bit in the bit map and then counting the bits below it in the map to calculate an index into an ordered sub-trie pointer table. The cost of null branches is thereby reduced to one bit. This method is not new and was first described by Bird [1977],Bird and TU. [1979], who conceived a hardware solution to this problem. An efficient software algorithm to count bits in the bit map provides the critical component for an AMT.

The fundamental problem is to derive a sub-trie pointer from the array bit map given a symbol. All the bits in the map do not need to be counted at search time. The high bits in the symbol are used to index directly to a word in the bit map. At insert time a tally of the bits in the map below this map word is computed. This is done by incrementing all tallies above the current one each time a new entry is made in the map. Then, at search time, only the bits in the specific map word need be counted and added to the offset calculated by corresponding tally. The result becomes the index into the sub-trie node table, figure 8.

The technique used to count the bit population is a variation on and a squeezed version of the algorithm to be found in *Hacker's Memory* programming tricks, used for Bit-Board manipulation in the Darkthought chess program. On several computer architectures the bit population count can be accomplished by a single

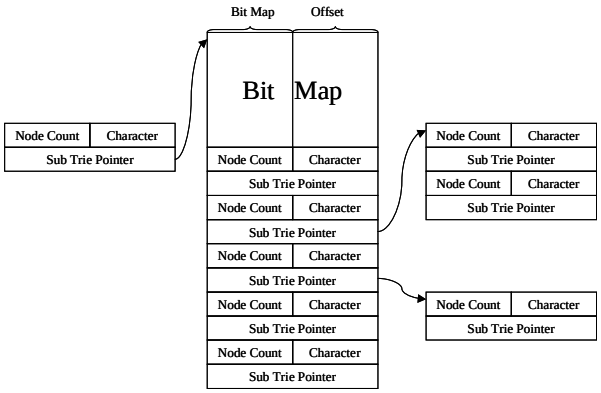

Fig. 9. An Array Mapped Tree.

CTPOP instruction and on these systems AMT performance will improve accordingly.

Two bits are needed to represent each possible symbol value, one for the map and one in the tally. Although small this still gives an overhead of 64 bytes per node for a 32 bit implementation and cardinality of 256. All nodes cannot be mapped this way. In order to reach an acceptable memory cost the typical node branch distribution is exploited to create a hybrid, illustrated in figure 9.

Nodes, with a branch count less than some value n, are not bit mapped but searched sequentially. Empirically $n = 5$ gives a good balance between space and speed. Each basic AMT node contains a 16 bit sub-trie node count, a 16 bit character value and the pointer to the next sub-trie. For nodes with only a few branches the valid branch characters are stored in the AMT node. Each branch list of nodes is kept in ascending sorted order. The node branch count distribution indicates that the majority of branches will be one or two. In these cases the linear search makes the same number of probes as a binary search yet the set up cost is lower. In the three and four cases, which represent a small minority, the linear probe is only one worse than the binary search. Overall the linear search wins. A data structure and the associated method to search such a tree with cardinality of 256 is shown in figure 10.

The critical step is the computation of the number of ones in the bitmap. Using the method shown in figure 11 this can be achieved in a small time, independent of bits set in the map. Although this computation looks lengthy it uses only shifts and adds, after compilation it becomes register based and is completed without memory references. Even so the CTPOP instruction would be better, in fact the entire search algorithm can be reduced to just 11 native Alpha instructions. Using the instruction set of the Intel x86 P3 the hybrid described here performs pleasingly well, search and insert costs are independent of the key set size and the space required is typically 8 bytes per node plus a 5% overhead for the bit maps.

```
class AMTNode {
unsigned int NodeCnt:16,Chr:16;
#ifdef AMTSMALL
union{
  AMTNode *IndexBaseA;
  int Value;
  };
#else
  AMTNode *IndexBaseA;
  int Value;
#endif
AMTNode *IndexBase(){return IndexBaseA;}
void SIndexBase(AMTNode *IB){IndexBaseA=IB;}
AMTNode(){IndexBaseA=NULL;NodeCnt=0;}
friend class CAMT;
};
```

Fig. 10. The AMT data structure.

4. UNARY SEARCH TREE (UST)

In the AMT a bitmap was used to select the appropriate sub-trie pointer. The size of this bitmap becomes the practical limit on allowing increased cardinality alphabets. A variation using the same basic node structure but replacing the bitmap search with a binary search of the pointer list and the UST *Next* method is illustrated in figure 12.

As the data structure uses just one pointer I followed the Bentley and Sedgewick convention and called this a Unary Search Tree. It is significantly faster than the TST and requires less memory space to represent the same tree. A UST has the same node structure as the AMT described above.

5. COMMON TREE ATTRIBUTES

All the trees described support the required important tree operations, since these are common across all the tree forms they are described here rather than with each tree.

5.1 Tree Tail Compression

For short keys the memory efficiency is reasonable. Many of the initial characters in keys share the same node. If there are 500 keys starting with the letter **A** then only one node is used to represent all 500. However, if the keys are longer than $log_s N$, where s is the symbol set size and N is the number of keys, then each additional key character takes one full node. This becomes very wasteful for even moderate length keys. For long keys this is costly, requiring at least 8 bytes per character in the above implementations. These expensive tails can easily be stored as a string.

As a new key is inserted it is converted to nodes until the point at which the key becomes unique. The remainder of the key is stored as a string. The case when a new key only becomes unique part way through a tail string requires that

```
inline AMTNode *CAMT::Next(AMTNode *pNode,unsigned int chr)
{
AMTNode *pList;
int L,H,M,NCnt;
if(!(NCnt=pNode->NodeCnt))return NULL;
pList=pNode->IndexBase();
if(NCnt<=BMLim){
  do{
  if(chr==pList->Chr)return pList;
  pList++;
  }while(--NCnt);
  return NULL;
 }
  unsigned int Idx,Map;
  Idx=chr>>5; //Get top bits
  chr&=0x1F; // clear top bits in chr
  Map=((AMTBMap *)pList)[Idx].BitMap;
  if(!(Map&(1<<chr)))return NULL; // no match
  Map&=(~((~0)<<chr));
  // Count Bits
  Map-=((Map>>1)&K5);
  Map=(Map&K3)+((Map>>2)&K3);
  Map=(Map&KF0)+((Map>>4)&KF0);
  Map+=Map>>8;
  return &pList[((Map+(Map>>16))&0x1F)
     +((AMTBMap *)pList)[Idx].Offset+LList];
}
```

Fig. 11. The *Next* function based on an AMT

the existing string must be converted into nodes up to that point and the new tail stored before adding the new key. With this little added complexity significant storage space can be saved and search times improved. For long keys the search becomes a simple string comparison.

With the ACT the tail can be packed into ACT nodes. This has the benefit that they too are easy to relocate and represent more *sand* in the knapsack. Timings for this variation are denoted by ACT(c) to be found in table 3 at the end of the paper.

The AMT(c) was implemented with an alternative method namely that tree nodes are only created when characters differ between two keys. The whole key is stored as a string, each tree node has an additional pointer to a character in the stored string and the character count to the next multi-way branch. In this way the nodes required per key stored is significantly reduced. Between branch nodes a fast memory byte compare tests characters in the stored string and against those in the key. A suitable *Find* method is illustrated in figure 13. In some applications this has the added advantage that the string pointer can be used as a *handle*. Note also that an AMT can be squeezed to use an average of about 5.5 bytes per node with

```
inline  USTNode *CUST::Next(USTNode *pNode,unsigned int chr)
{
USTNode *pList;
int L,H,M;
if(!pNode->NodeCnt)return NULL;
pList=pNode->IndexBase();
L=0;H=pNode->NodeCnt-1;
  do{
    M=(L+H)/2;
    if(chr==pList[M].Chr)return &pList[M];
    if(chr>pList[M].Chr)L=M+1;
    else H=M-1;
    }while(L<=H);
  return NULL;
}
```

Fig. 12. The *Next* function based on a UST

```
int CAMT::Find(CString str)
{
   AMTNode *P,*C;
   const unsigned char *s,*ps;
   int Cnt;
   if(*(s=((unsigned char *)(LPCTSTR)str))==0)return 0;
   P = Root;
   ps=s;
   while(C=Next(P,*s)){
     s++;
     if(Cnt=C->ChrCnt){
       if (memcmp(C->Key+(s-ps),s,Cnt))return 0;
       else s+=Cnt;
       }
     P=C;
     }
   if(*s==0)return P->Value;
   return 0;
}
```

Fig. 13. *Find* function for AMT(c) with external strings.

a small loss of speed for ultimate space efficiency. Both of these implementations use less space than Hash Trees and perform faster searches.

A Directed Acyclic Word Graph (DAWG) may be considered for further savings in some applications, Appel and Jacobson [1988].

5.2 Deletion

Deleting a key from the AMT and ACT is low cost. The key entry is found using the standard lookup routine, where the key is a sub-string of another key, then all that is required is the resetting of *Term* to false or removing the *ghost* node.

If the key is a unique long key then a little more work is required. The tree is traversed starting from the leaf node to the multi-way branch node that leads to this leaf node, typically a traverse of one or two nodes. Then this branch is clipped from the trie by removing its entry. It and the following nodes are returned to the free node pool. Deletion is only slightly slower than lookup and faster than insertion.

5.3 Sequential Key Access

Many applications need to be able to access keys from a dictionary in a sorted sequential order. The ordered data structure in AMT and the linked list structure in each sub-trie of an ACT enables simple traversal and a very fast iterator can be easily coded.

5.4 Query Set Definition and Regular Expressions

Frequently search queries need to be qualified by filters, examples could be ignoring Upper and Lower case sensitivity, abbreviations or using the regular expression syntax **.** and ***** to define a search key pattern. **BB..d*** means return in sequence all keys starting with **B** in the first two positions, any symbol in the third or fourth position, **d** in the fifth and any number of any symbols to follow. Rivest [1976] and Bentley and Sedgewick [1997] for a detailed description and analysis.

In the first example, keys containing both upper and lower case characters may have been stored in the Tree in the normal way. At query time all the keys matching a query key need to be found irrespective of case. A slightly modified lookup function achieves this. In the standard lookup algorithm a character value is used to index into a trie. Here the character value and its other case value are used. If the test is successful the test continues to the next character until a failure is found. At this point it is necessary to back track to the last successful branch on case and try the alternative. In general it could take $2N-1$ steps to test the key. However, if the test is biased to the most dominant case e.g. lower case for English text, then only a few steps are needed in practice.

In some applications such as a command line interpreter you may wish to allow unique abbreviations of the commands or in a text editor give intelligent type ahead. For the potential abbreviated key the tree is searched until there are no more multi-branches and the key is uniquely defined. The remainder of the key from this point onwards becomes the type ahead characters. Further more this comparison can be made as keys are typed allowing the command line interpreter to fill in the full command.

The third case is dealt with in much the same way as the first. When a **.** is encountered all the entries for the corresponding trie are returned in sequence. When the ***** is encountered it is treated as any number of **.** to the right of its position. When specified letters are encountered they must match exactly.

5.5 Priority Queues

Priority queues are important structures used to solve many everyday application problems. They appear in task scheduling, graph searches and many other common algorithms. Here a Map is required that enables keys to be inserted in a random order using a key that defines the priority of an entry and then is systematically retrieved by the highest priority key. The key could be for example the schedule date and time for events to occur. These may be inserted at any time. The scheduler monitors current time and retrieves tasks from the top of the scheduling queue to execute as time passes. This could be at the level of an operating system scheduling programs or a manufacturing process scheduling work.

A priority queue needs to support several operations namely insertion, deletion, find first, find last, replace and joins, on keys in an efficient way.

For almost all key types an ACT (or AMT) can be used to implement priority queues. Insertion, head removal and deletion can all be fast functions independent of the key set size. Joins, the merging of two queues, can be performed efficiently too.

Insertion and deletion are carried out as described earlier. The retrieval of the top or first key is straight forward. Starting at the root node the tree is traversed always picking the lowest branch value in a trie. Since this is carried in the data structure as *FirstChar* this is rapidly done. Typically only $log_s N$ nodes need to be traversed, where N is the number of keys and s is the symbol set cardinality. As discovered previously, keys are typically unique after this number of symbols in the key. Alternatively a head of queue pointer may be maintained and updated on insertions.

In practice an ACT and AMT perform extremely well as a priority queue and increasing free memory in an ACT ensures a consistently fast insert.

5.6 Range Searching

All the Trees support fast range searching. This is the problem of finding the number of keys that lie between an upper and lower key defining the range. This may be a set of names, co-ordinates or values. Adding a range data element to each node allows ranges to be calculated quickly. The range element is set to the count of keys below that node in the tree.

To find the number of keys in a given range the lower key is found in the Tree. The first character is taken and used to index into the root trie. The range count is stored. The range count for characters found early in the linked list is added to this, then as each subsequent character is used to index into their respective tries the range count of characters earlier in the character link list of that trie are added to the previously calculated range count. Typically there are few additions to make. The process is repeated using the higher bound key. The range is given by subtracting the lower bound result from the upper bound result.

Maintaining the Range information in the tree only requires incrementing the data element as new keys are added or reducing the count when keys are deleted.

6. PERFORMANCE COMPARISON

6.1 Test Method

All the algorithms were tested with the same wide variety of unique key sets with different alphabet cardinalities. The sets were in random, sorted order or semi-ordered sequence. The test sets were produced using a custom pseudo random number generator. These were created by first generating a random prime number which was repetitively added to an integer to produce a sequence of 32 bit integer keys. For high value prime numbers this sequence is pseudo random while for low value prime numbers the sequence is ordered. Values in between generate semi-ordered sequences.

A random cardinality value was generated in the range 20 to 255 and used as a radix to convert the integer key to a string of characters. Additional characters were added to create an eight-character key. All the tests reported in Table 3 were for 8 character key sets created in this manner.

The technique ensures unique key production and allows search test key sets to be generated with different sequences to those used during insertion yet from the same key set.

AMT(c) and ACT(c) indicate the tail string compressed version of the algorithms. The initial hash table size was set to 50000 and the results reflect the load factor around this value. The TST algorithm used is an enhanced version that balances sub-trie trees.

Each algorithm's performance was measured on an Intel P2, 400 MHz with 512Mb of memory, NT4 and VC6, they are compared in Table (3). Times shown are in μS per insert or search for 8 character keys averaged across 2000 test key sets.

Insert and search times were measured using the QueryPerformanceCounter() function.

6.2 Results

Table 3. Comparative performance of search trees.

Algo.	**SetSize**	**10K**	**20K**	**40K**	**80K**	**160K**	**320K**	**640K**	**1280K**
ACT(c)	Insert	5.03	6.13	9.00	14.49	17.27	14.94	14.66	14.79
	Search	1.13	1.24	1.37	1.51	1.52	1.59	1.66	1.74
AMT(c)	Insert	6.36	7.03	8.39	12.20	13.45	12.55	12.57	11.11
	Search	1.37	1.57	1.68	1.77	1.90	2.00	2.14	2.29
Hash	Insert	2.97	2.74	2.90	3.24	3.65	4.56	6.24	9.64
	Search	1.29	1.49	1.88	1.97	2.48	3.40	5.15	8.54
TST*	Insert	10.23	11.08	12.67	14.72	14.59	13.76	13.07	12.26
	Search	2.78	2.99	3.27	3.57	3.87	4.16	4.51	4.86
ACT	Insert	6.12	7.54	10.12	12.89	14.4	20.9	26.6	21.26
	Search	1.69	1.95	2.09	2.28	2.22	2.26	2.31	2.38
AMT	Insert	10.06	11.02	11.02	12.85	13.40	12.57	11.98	11.53
	Search	1.87	1.97	2.09	2.13	2.20	2.31	2.46	2.55
UST	Insert	9.81	10.08	10.67	12.41	13.10	12.40	11.86	11.45
	Search	2.18	2.35	2.43	2.57	2.75	2.90	3.06	3.22

7. CONCLUSIONS

As can be seen from the comparison UST, ACT and AMT all give an excellent performance, are space efficient and support sorted order retrieval. Both an AMT and ACT give better search performance than Hash Tables yet use less space and are extensible. They perform particularly well in mostly-not-found applications.

For medium sized key sets both TST and UST out perform the logarithmic performance anticipated by Clement, Flajolet, and Vallee [1998]. This is attributed to the cache effect described above. The critical parameter is the number of non-cache resident memory references made in the algorithm. The trie with most branches is typically traversed most often and these nodes tend to be cache resident therefore not attracting the additional time required to complete a binary search of the TST or UST trie nodes. However, for very large key sets or for applications with mixes of lookups and other computation the advantage of an ACT or an AMT becomes apparent. Notice as the set size grows for a TST or UST, the reducing cache hits increases search times. A UST has the same lgN cost as a TST yet occupies 60% less space and is faster. The constant search cost, independent of set key size of an ACT or AMT together with their space efficiency makes them particularly attractive for very large key sets.

An ACT performs well for typical key sets but insert performance can deteriorate seriously with some key sets. Once the keys no longer sparsely populate the space of all possible keys the trie placement becomes difficult and memory use accelerates. However, the ACT provides an excellent supporting data structure for FSA's and symbol tables.

An AMT has an excellent performance independent of key sets, compact, simple to code and robust. Implementation on a Compaq Alpha processor, Motorola Power PC, Sun UltraSparc or Cray using CTPOP provides an attractive solution for any high performance search problem. They are faster and more space efficient than Hash Tables, yet are extensible and support sorted order functions.

Finally, its worth noting that **case** statements could be implemented using an adaptation of the AMT to give space efficient, optimized machine code for fast performance in sparse multi-way program switches.

ACKNOWLEDGMENTS

I would like to thank Prof. Martin Odersky, Christoph Zenger and Mathias Zenger at the Labo. Des Methodes de Programmation (LAMP), EPFL, Switzerland for their review of the draft paper and valuable comments.

REFERENCES

Appel, A. and Jacobson, G. 1988. The world's fastest scrabble program. *Communications of the ACM 31*, 5 (May), 572–578.

Bentley, J., McIlory, M., and Knuth, D. 1986. Programming pearls: A literate program. *Communications of the ACM 29*, 6 (June), 471–483.

Bentley, J. and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (1997), SIAM Press* (1997).

Bird, R. 1977. Two-dimensional pattern matching. In *Inform Procee. Lett: 6(5)* (1977).

Bird, R. and TU., J. 1979. Associative crosspoint processor system. *U. S. Patent 4,152,*

762.

Brandais, R. 1959. File searching using variable length keys. In *Proceedings of Western Joint Computer Conference*, Volume 15 (1959), pp. 295–298.

Brown, D., Baker, B., and Katseff. 1982. Lower bounds for on-line two-dimensional packing algorithms. *Acta Informatica 18*, 207–225.

Clement, J., Flajolet, P., and Vallee, B. 1998. The analysis of hybrid trie structures. In *Proceedings of Western Joint Computer Conference*, Volume Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (1998), pp. 531–539. SIAM Press.

Fredkin, E. 1960. Trie memory. *Communications of the ACM 3*, 490–499.

Knuth, D. 1998. *The Art of Computer Programming, volume 3: Sorting and Searching, 2nd Ed.* Addison-Wesley, Reading, MA.

Rivest, R. 1976. Partial-match retrieval algorithms. *SIAM Journal on Computing 5*, 1, 19–50.